**KRF**

**Knowledge Representation Framework Project**
Department of Computer and Information Science, Linköping University,
and Unit for Scientific Information and Learning, KTH, Stockholm

Erik Sandewall

# Software Architectures and Languages for Autonomous Intelligent Agents

*Version I, Incomplete*

# Introduction and Prerequisites

The incorporation of intelligence in *autonomous software agents* is a central area of interest for Artificial Intelligence, for two reasons: it is required in a number of application areas, and it is a generative topic so that research in this area produces techniques that turn out to be useful for other types of applications as well. The present compendium is an introduction to contemporary *software techniques* in this area, that is, software architectures, representation languages and their associated software "engines," and so forth.

This compendium uses the notation and terminology of the Knowledge Representation Framework, and it is recommended to study the following texts before the present one:

- Knowledge Representation Framework: Overview of Languages and Mechanisms
- Compendium of Programming Techniques for Knowledge-based Autonomous Systems, Part I: List Processing
- For the reader who is not already familiar with formal logic, in particular first-order predicate logic: Notes on Propositional and Predicate Logic for the Knowledge Representation Framework.
- The Goals of Artificial Intelligence Research - A Brief Introduction

All of these are available on the webpage of the Knowledge Representation Framework Courseware, `http://www.ida.liu.se/ext/krf-courseware/` .

In the sequel we shall simply write "intelligent agents" and let the attribute "autonomous" be understood.

## About Cited Text

This compendium makes fairly extensive use of clippings from webpages and reports that describe particular systems and languages. This is not done merely for the convenience of the author. I believe that often the best concise description of a system is made by the people that built it, and if such a good description is available then it does not make any sense to reformulate it merely for the sake of originality. On the other hand it may sometimes be necessary to make minor adjustments in cited text fragments in order to integrate them well into the surrounding text. Therefore, when sections of text in this compendium are marked as being citations from an external source, it shall always be understood that the citation may differ in minor ways from the original, but without changing its essential meaning.

Citations from Wikipedia are made in accordance with the requirements that are stated on the Wikipedia website, with direct links to the Wikipedia page being cited. In other cases, and except for very short citations and one case where the original author could not be reached, I have requested and obtained permission from the authors or proprietors of the original text to use it here, and these have also seen the modifications.

# Chapter 1

# Intelligent Agent Architectures and Terminology

The overall structure of a piece of software is often called its “architecture” or its *software model.* The architecture specifies the answers to questions such as, for example,

- What are the major collections of information in the system, and how is information exchanged between them?
- What are the major computational processes in the system, and how are they invoked, how do they invoke each other, and more generally what are the methods for controlling the overall flow of information?
- What are the notational conventions for information when it is stored in the system, and when it is exchanged with users and with other systems?

When we think of intelligent agents as a particular kind of software system, and a particular topic of research and development, there is an assumption by many that there can be a specific software architecture or software model that is appropriate for intelligent agents in general, although of course particular applications will require particular variations of the general design.

The present chapter will focus on one particular intelligent agent architecture, the *Belief-Desire-Intention Software Model,* or the *the BDI model* for short. This model is widely accepted in Artificial Intelligence in the sense that there are many interpretations and variants of it, so many researchers use it although they also make adaptations to fit their specific experience and preferences. Also, even those who favor other software models than this one will anyway often describe their approach by how it differs from the BDI model.

## 1.1 Components of the BDI System State

Many computer programs are *request-driven* in the sense that their basic behavior is to wait for a command or request from a user, execute the command or service the request, and then wait for the next one. This applies in particular for many softwares that people use in their personal workstations or laptops, such as systems for document preparation and electronic mail, but it also applies for servers and for transaction processing systems for example in banking.

Autonomous agent systems have a fundamentally different character. They are characterized by a *system state* that contains the agent's "state of mind" at each point in time, and by a cyclic process that uses and updates the system state repeatedly. Each *main cycle* of the agent's behavior may receive information from sensors, if we are talking about a physical robot, as well as from interactions with users and with other agents. However, what the agent does then is primarily determined by its current *intentions* which is one part of its system state. Its intentions may be revised due to the inputs it has received, of course, but such a change of intention is only done after due deliberation and not immediately in order to service a particular request.

The structure of the system state is therefore of paramount importance, and the BDI model begins by defining and naming a number of system state components. The following list of components is taken from the Wikipedia article on the Belief-Desire-Intention Software Model [1] while just omitting a few phrases that are of marginal importance at this point. It is representative for the literature although different authors differ somewhat in their use of the terms. The definitions below will therefore be used for the present compendium and the present course. Words and phrases that are written in italic style will be defined later on.

**Beliefs:** Beliefs represent the informational state of the agent, in other words its beliefs about the world (including itself and other agents). Beliefs can also include *inference rules,* allowing *forward chaining* to lead to new beliefs. Using the term belief rather than knowledge recognizes that what an agent believes may not necessarily be true (and in fact may change in the future).

**Beliefset:** Beliefs are stored in a database (sometimes called a belief base or a belief set), although that is an implementation decision. [*We will use the term beliefbase rather than beliefset*] .

**Desires:** Desires represent the motivational state of the agent. They represent objectives or situations that the agent would like to accomplish or bring about. Examples of desires might be: find the best price, go to the party or become rich.

**Goals:** A goal is a desire that has been adopted for active pursuit by the agent. Usage of the term 'goals' adds the further restriction that the set of active desires (i.e., current goals) must be *consistent.* For example, one should not have concurrent goals to go to a party and to stay at home - even though they could both be desirable.

**Intentions:** Intentions represent the *deliberative state* of the agent - what

[1]http://en.wikipedia.org/wiki/Belief-Desire-Intention_software_model

the agent has chosen to do. Intentions are desires to which the agent has to some extent committed. In implemented systems, this means the agent has begun executing a plan.

**Plans:** Plans are sequences of actions (recipes or knowledge areas) that an agent can perform to achieve one or more of its intentions. Plans may include other plans: my plan to go for a drive may include a plan to find my car keys.

**Events:** These are triggers for *reactive activity* by the agent. An event may update beliefs, trigger plans or modify goals. Events may be generated externally and received by sensors or integrated systems. Additionally, events may be generated internally to trigger decoupled updates or plans of activity. *End of quotation.*

Desires can include both *standing desires* such as “do not allow any malware into the computer” and *occasional desires* that apply at a particular time, such as “obtain printouts of all the pictures in the favorite photos directory.” The same distinction applies for goals. Intentions are always occasional, on the other hand, although they may involve repeated execution of a particular task at several points in time, for example “backup this USB every morning.”

The elementary components of a plan are called *actions,* so each plan is either an action or a partially ordered set of *plan steps;* each plan step is either an action or a subordinate plan, called a *subplan.* A plan consisting only of actions and no subplan is called *sequential* iff the order on the plan steps is total. An arbitrary plan is called sequential iff the order on the plan steps is total and all its subplans are sequential. Some systems and approaches include additional information in plans, for example, information about the timing and duration of actions and subplans.

An action consists of an *action verb* and *arguments* for the action. Actions may be *primitive* or *compound* according to the action verb. A compound action verb is defined by a *plan script* that shall be performed in order to perform an action with that verb. A plan script is like a plan except that it can refer to, and make use of the arguments of the action invoking the script. A primitive action verb must instead have an attached procedure in a conventional programming language that is executed in order to perform an action with that verb.

A plan that consists of a single, primitive action is called *atomic;* plans consisting of several actions or subplans, or of a single, compound action are called *composite* or *non-atomic.*

## 1.2 The Main Cycle of the BDI Model

### 1.2.1 The Generic BDI Main Cycle

The following is the simplest version of the main cycle in the BDI model. This cycle is defined as a *script,* namely, as a simple program consisting of a few steps, each of which must be further defined, and which can be defined differently in different application situations. This segment of text is quoted from the important article by Georgeff and Rao in 1995 [2] where they describe the principles for their Procedural Reasoning System (PRS) and its successors. This article is recommended reading. We quote a passage of its text, with a few explanations inserted.

```
initialize-state
repeat
   options := option-generator(event-queue)
   selected-options := deliberate(options)
   update-intentions(selected-options)
   execute()
   get-new-external-events()
   drop-unsuccessful-attitudes()
   drop-impossible-attitudes()
end repeat
```

The value of the variable `event-queue` is maintained throughout the process and is initially an empty set or queue. At the beginning of every cycle, the option generator reads the event queue and returns a list of options, that is, things that may become intentions. Next, the deliberator selects a subset of the options to be adopted and adds these to the intention structure. If there is an intention to perform an elementary action at this point in time, then the agent executes it. Any external events that have occurred during the interpreter cycle are then added to the event queue. Internal events are added as they occur. Next, the agent modifies the intention and desire structures by dropping all successful desires and satisfied intentions, as well as impossible desires and unrealizable intentions. *End of quotation.*

The plan library is fixed in the basic model and there is no mention of how to generate new plans. Therefore, if you consider three major activities for an autonomous intelligent agent, namely:

- Create plans
- Choose between possible plans
- Execute plans

then the main cycle script separates the latter two activities. However, the script does not say anything about the activity of plan creation. It is organized around some of the presumed BDI system state components that were defined above, namely, events, intentions, and plans. One must assume that several of the operations such as `deliberate` and `option-generator` make use of the agent's beliefbase. The notions of *desires* and *goals* are only present in this main cycle in the sense that "attitudes" is used as a common name for desires, goals and intentions in the script. Furthermore, selection of intentions and plans may take standing desires and goals into account in order to increase or decrease the estimated merit of adopting that intention or plan.

[2]Rao, M. P. Georgeff. (1995). "BDI-agents: From Theory to Practice". Proceedings of the First International Conference on Multiagent Systems (ICMAS'95). https://www.aaai.org/Papers/ICMAS/1995/ICMAS95-042.pdf

### 1.2.2 Choice of Intention and Plan

The operation `option-generator` in the main cycle script depends on the application and has to be defined in its own way for each of them. We therefore proceed to discussing the next two operations in the script, that is, `deliberate` and `update-intentions.`

Going back to the definitions, intentions are "desires to which the agent has to some extent committed" and a plan is "a sequence of actions." In a simple interpretation of this, an intention may be for example "go to the train station" and a plan may be "walk to the bus stop, wait for the first bus to the train station, go with that bus." The agent may contain a *plan library* consisting of fixed plans, and with information about which plans are appropriate for which intention. When performing the main cycle script it needs to look up available plans for a selected intention, choose between them, and use one for the step called `execute.`

At some points the agent has to make a choice between different possible intentions, such as "go to the train station" or "cancel the train trip." This choice should be based on the desires and goals that the agent has established in its system state. The definition of the main cycle script is not clear about whether the choice of intention, and the choice of plan for a given intention shall be included in `deliberate,` in `update-intentions,` or even in `execute.`

The design choice in this respect depends on whether the choice of intention and the choice of plan are interdependent or not. Suppose the agent has the option of "going to Stockholm by train over the day" or "cancelling the train trip." It selects the former option, i.e. turns it into an intention, based on the fact that it is in line with some of its goals, such as seeing football (soccer) matches. Then it considers the possible plans for this. The first step of the plan will be going to the train station, and there are several subplans for this in the plan library, including walking, taking the bus, and taking a taxi. Suppose now the agent considers all the plans, decides that each of them has significant disadvantages, and then changes the intention to the second one and cancels the trip. How is this possible in the framework of the main BDI cycle?

One possibility is to arrange that choice of intention and choice of plan are done in sequence. Then the choice of going to Stockholm is done in the `deliberate` step, and the `update-intentions` step consists of extending the structure of selected intentions by adding information to them, in particular, adding an appropriate plan for the selected intention. If no plan can be found then the intention *fails* in the execute step, so nothing happens in this respect, and it is removed in the operation `drop-impossible-attitudes.`

Another possible realization of the main cycle script is to let the `deliberate` step consist of more things: consider the newly generated options, identify possible plans for each of them, and evaluate each of them with respect to both the importance of the intention (i.e. how it relates to one's desires and goals) and the quality of the considered plan with respect to those desires and goals. Then `update-intentions` consists of making a choice between the intention/plan pairs based on this evaluation.

The main cycle script shown above can therefore be understood and implemented in more than one way. When one is actually designing the software for an intelligent agent one may use it as a first outline of the design in a top-down design process, but it must be complemented by a number of additional design decisions.

### 1.2.3 Alternative Models for Plan Execution

Although the main cycle script separates choice of plans from execution of plans, it actually only provides structure for the former activity, since plan execution is represented by the simple operation `execute.` The exact character of plans, the method for executing plans, and the relation between intentions and plans must therefore be clarified, and these questions may be answered differently in different uses of the main cycle script.

Consider therefore the situation where the agent has committed to one particular intention and one particular, non-atomic plan for realizing that intention. Assume for simplicity that the plan is also sequential. One way of defining `execute` is to say that it will perform the entire plan in one go, so that the `execute` operation has finished when all the steps in the plan have been performed, and then the main cycle script can proceed to its next step.

This is an appropriate definition in some cases, but not always. It is problematic if the plan execution takes a certain time, as is often the case for mobile robots, for example, and for two reasons: first, it precludes any activity with the other steps in the main cycle script while the robot is performing its actions, such as moving from one place to another, and secondly it does not facilitate the handling of problems that may arise during the execution of the plan, namely, if something should go wrong then.

An alternative arrangement is to let the interpreter for the main cycle script maintain the execution stack for each plan being performed, so that the `execute` operation means "advance one step in the current plan, perform one more action, but no more." This has the effect that all the other steps in the main cycle script are executed once again between each step in the plan and its subplans on all levels. The definition in Subsectin 1.2.1 favors this approach. This solves the mentioned problem but introduces another one, namely, that it may slow down the system considerably. We shall refer to this as *BDI-aware* plan execution.

A third possibility is to use multiple *execution threads* as provided by the operating system or the programming language at hand, so that the execution of a plan can continue in a separate thread and the main cycle script can proceed in the original thread. In this case the definition of `execute` in the script shall really be understood as `invoke` since the only thing that happens before the script proceeds to its next step is that the execution of a plan is *started.*

The separate-thread approach has several advantages, but it is less easy to implement and it requires certain extensions to the basic model in order to handle the situations that arise when the execution of a plan *fails* without having achieved the intention that the plan was started for. The failure of the plan should then generate an event, in the sense of that term in the BDI model, in order to communicate the failure back to the main cycle. This event should be taken care of there by a reconsideration of the agent's intentions, for example, cancelling the intention in question since it could not be realized, or retaining the intention and trying again with another plan.

A similar thing should in fact be possible in the system even if a plan proceeds according to its definition but the `execute` operation recognizes that the operation takes unexpectedly much time or other resources. In such cases should the plan also generate a warning event that can be picked up in the main cycle, and that may cause it to discontinue the execution of the plan in favor of some other action.

### 1.2.4 Restrictions on the BDI Model

As Georgeff and Rao write in their article about the PRS system, the main cycle script and its description is an idealized definition that corresponds well to the underlying psychological theory, but it is not a practical system for rational reasoning. The authors propose the following restrictions in order to obtain reasonable performance and a manageable system:

- Consider only beliefs about the current state of the world. Each belief is a literal in the sense of logic, that is, an expression consisting of a predicate and its arguments, or the negation of such an expression.
- Represent information about the means of achieving certain future world states as a library of *plans.* Each plan has a *body,* an *invocation condition* that specifies what intentions the plan in question may achieve, and a *precondition* that specifies what must hold in order for the plan to be executable.
- The system forms an intention by adopting a plan, and in each case it creates a separate process for executing that plan, allowing for the plan to contain invocation of subplans.

### 1.2.5 Logicist Realization of the Main BDI Cycle

It may seem strange at first that the main BDI cycle defines a somewhat elaborate structure for the selection and management of intentions and plans, but it says absolutely nothing about the execution of plans. One reason for this is however that there are efforts to define the operation of the steps in the main cycle in terms of logic, except usually the `execute` step, and not merely as a conventional subprogram. Such a logicist definition of the main cycle may be used as a *specification* for conventional programming, or it may be used *directly* by manipulating the logic formulas in the operation of the intellligent agent.

*The following also taken from the Wikipedia page; should be simplified.* An important aspect of the BDI software model (in terms of its research relevance) is the existence of logical models through which it is possible to define and reason about BDI agents. Research in this area has led, for example, to the axiomatization of some BDI implementations, as well as to formal logical descriptions such as Anand Rao and Michael Georgeff's BDICTL. The latter combines a multiple-modal logic (with modalities representing beliefs, desires and intentions) with the temporal logic CTL*. More recently, Michael Wooldridge has extended BDICTL to define LORA (the Logic Of Rational Agents), by incorporating an action logic. In principle, LORA allows reasoning not only about individual agents, but also about communication and other interaction in a multi-agent system. *End of quotation.*

## 1.3 The SOAR Architecture

Although the Belief-Desire-Intention Software Model is widely used and adapted, it is not the only available one. For perspective we shall briefly describe the SOAR architecture [3] which actually predates the BDI model.

The SOAR Architecture was proposed by John Laird, Allen Newell and Paul Rosenbloom in the mid 1980's as a step towards a "Universal Theory of Cognition." In other words, it was motivated at least as much by the discipline of Psychology as by Artificial Intelligence. It proposes that cognitive behavior shall be modelled using the following main cycle:

- Input
- Elaboration
- Decision
- Application
- Output

In the input step the system receives observations from its sensors, in a broad sense of that word, so for example the reception of a command phrase from the user is considered as just one of those inputs. The result of the input step is to update the agent's beliefbase, including in particular its model of the environment where it is operating. The elaboration step consists of drawing conclusions from the state of the beliefbase that resulted from the input step. These "conclusions" may be bona fide conclusions about the current state of the outside world, but there can also be *internal conclusions.* One particular kind of internal conclusions is *suggestions,* that is, expressions representing some action that it might be appropriate for the agent perform. Suggestions are similar to 'options' in the BDI model.

The purpose of the decision step is to select one action from the suggestions in the outcome of the elaboration step. In simple cases there is just one suggestion and this suggestion is a good one, according to criteria being used by the decision step, but in many cases this is not so. Maybe there are several suggestions, or no suggestion, or one suggestion but one that does not meet the standards. Situations such as those are called *impasses* . In these cases the decision step starts a sub-process of the same form as the main process, with the goal of resolving the impasse and coming up with a result from the top-level decision step.

The application step executes the outcome of the decision step, and the output step reports the results to the parties concerned, for example, to a log that keeps track of past actions and their results.

[3]http://sitemaker.umich.edu/soar/home

A major difference between the BDI model and the SOAR model is therefore that the SOAR model cycle is defined to be *recursive:* at certain points it calls itself recursively and causes a sub-loop of the same kind as the loop at hand, although normally one with limited duration. There is no counterpart of this design choice in the BDI model. However, the same effect can be achieved there as well through a suitable definition of the intention structure, including in particular the use of a stack-like construct that replicates the recursive invocation of the main cycle in the SOAR model.

## 1.4 BDI-Aware Plan Execution

Let us return now to the alternative of BDI-aware plan execution which was mentioned briefly above, where the main BDI cycle proceeds step by step through the plan and all its subplans possibly on several levels, and where the revision of the intention structure occurs before every such step. This is done down to the level of primitive actions, and these are executed by either of the other two methods that were described in Section 1.2.3, that is, either by direct execution or by the use of concurrent execution threads.

In particular, consider the combination of BDI-aware plan execution with the use of *event-to-plan shortcuts* where the arrival of a particular type of event in `event-queue` is allowed to bypass the operations called `option-generator` and `deliberate,` and to proceed directly to the `execute` operation. This may be used for implementing *reflex actions* where a particular stimulus leads to an immediate and rapid reaction, for example in order to avoid danger or harm.

### 1.4.1 Understanding of Natural-Language Phrases

However, event-to-plan shortcuts can also be used for defining various kinds of standard behavior in the main cycle. Consider for example the case where the agent is able to receive phrases in natural language (English, Swedish, and so forth) and to react properly to them. This task is divided into two main parts: *understand* the phrase being received, and *respond* to it. Both of these steps may make use of the machinery of the top cycle: one may need to deliberate both in order to understand the phrase, and in order to answer it if it is a question, or to perform the requested action if this is what the phrase is about.

One way of implementing this assumes that there is one specific event type for the arrival of an incoming phrase, and each event of this type has the phrase in question as a parameter. One can then let the plan library include a plan consisting of the two actions or subplans for understanding the phrase and for responding to it, and where the first action produces a representation of the contents of the phrase in a representation that the second action can interpret. The event type for arrival of incoming phrase obtains a shortcut to this plan. After this, each phrase arrival leads to an immediate start of the phrase understanding action, and when it has been completed successfully it is followed by the response step.

This way of organizing the computation has the advantage that the treatment of natural-language input is well integrated with other causes that lead the agent to perform actions, such as its own spontaneous reaction to events in the environment. It also means that failure to understand a phrase, or to perform a phrase as required, can be handled by the same failure-handling mechanisms as are used for other purposes in the main cycle.

What has been described is one example of a *cognitive plan,* that is, a plan consisting of actions that are performed internally in the agent itself, with effects on its system state, but without any significant effects on the agent's environment. Cognitive plans are a powerful means of augmenting the main cycle with facilities that are generally considered as important aspects of intelligence. We shall consider a few of those in the next subsection.

### 1.4.2 Cognitive Plans

*Key points in the intended contents in this subsection:*

- **Planning:** If the `deliberate` operation is not able to find a plan in the plan library for a selected intention, then it can generate a failure event that has a shortcut to the planning action, that is, the action of constructing a plan for achieving that intention
- **Case-Based Learning:** This cognitive action presupposes that the agent contains a *review action* that is invoked at the end of the activities for an intention, and that does two things: it accumulates the information about how the intention was processed into the agent's memory, and it analyzes whether there is something to be learnt from how this was done. This is an example of a cognitive plan. In order to have a loose coupling between a given action and the review of it, the things are set up so that the completion of an ordinary action generates an event of a particular type, which in turn may trigger the review action via the `option-generator` action, or even via a shortcut.
- **Error Recovery and Robustness:** When an action or plan fails, it may sometimes be necessary to do a certain amount of deliberation in order to figure out what were the causes of the failure, and to decide what to do next. Deciding what to do next is a natural task for the main cycle as such, but the *diagnosis* of what went wrong is an activity that has its own character and for which there exist systematic methods. It is therefore reasonable to consider diagnosis in this sense as a cognitive action in the agent.

In general, since the main BDI cycle script is expressed in such general terms, there are many needs for making it more specific, or to replace it with something else, in order to have something that is applicable for each type of application. However, in some cases it is possible to accomodate such needs by introducing a cognitive plan and to provide it with an appropriate definition, as these examples have showed.

# Chapter 2

# Systems for Planning and Plan Execution

As already discussed, the BDI models focuses on the mechanism for the choice of actions and plans in a deliberative agent, and it says little about planning (i.e. the making and the revision of plans) and about the execution of plans. A full description of an intelligent agent that is built according to the BDI architecture must therefore also include an account of how plan execution is performed in such a system.

The question of planning is more open: there are plenty of actual AI applications where there is no need for a separate planning component, so that the systems for these applications manage well with a flexible and robust plan execution component. However, in those cases where planning is required, it is a third topic that is closely interdependent both with plan execution, and with the questions of goals and intentions that are addressed by the BDI model.

In fact, there are also systems where the plan execution component dominates, and where questions of goals and intentions are handled in a limited way as part of the plan execution process. In those cases there is no significant BDI aspect in the system as a whole.

The present chapter will describe the major alternatives in these respects, beginning with planning and plan execution systems that have been developed together with the BDI model, and proceeding with two types of systems where the planning and plan execution aspect dominates.

## 2.1 BDI-based Task Execution Systems

### 2.1.1 Procedural Reasoning Systems

The original Procedural Reasoning System (PRS) was developed at Artificial Intelligence Center at SRI during the 1980s. It has been succeeded by a number of extensions and later developments, including SRI's SPARK system (SRI Procedural Agent Realization Kit) and the OpenPRS system

[1] both of which are presently in active use for major projects.

*Add more text here.*

### 2.1.2 AgentSpeak and Jason

*The following citation is from Lambda the Ultimate Weblog, [2] which also contains some discussion about AgentSpeak.* (check with ehud.lamm at gmail.com)

AgentSpeak is a Prolog-like resolution-based language, but which is extended to support agent-based programming in several ways, most importantly:

1. It extends the language, so that clauses can talk about not just satisfaction of predicates, but also of an agent desiring to bring about a predicate, and desiring to find out whether a predicate is true; and to distinguish between normal goals and special goals relevant to the BDI model (Belief-Desire-Intention model);

2. It amends the resolution engine to support what Rao calls reactive concurrency, where agents form plans via a process resembling SLD-resolution, but plans are formed or abandoned on the basis of agent-internal reactions called triggering events. *(End of quotation)*

One should not confuse AgentSpeak in this sense with a speech generation product with the same name.

AgentSpeak should be thought of as a prototype for demonstration purposes, rather than as a platform for practical systems. It has been further developed into the Jason system [3]

## 2.2 Hierarchical Task Networks

A major approach for planning and plan execution systems is *Hierarchical Task Networks,* HTN. A task network is a data-structure representation of a plan, consisting of actions and subplans as defined in Section 1.1, and where the properties of the plan's components and the dependencies between them have been made explicit using links in a network structure. Dependencies may include the requirement that one plan step shall precede another plan step since the postcondition of the former achieves the precondition of the latter. Dependencies may also include metric temporal constraints having to do with the durations of various plan steps, or the permitted window in time for the start or the end of some plan steps or the entire plan.

The following are some implemented HTN systems:

- Nonlin, one of the first HTN planning systems, developed at the University of Edinburgh, UK

[1] https://softs.laas.fr/openrobots/wiki/openprs
[2] http://lambda-the-ultimate.org/node/3003
[3] http://dtai.cs.kuleuven.be/projects/ALP/newsletter/aug06/nav/articles/article5/article.html

- O-Plan, developed at the AIAI at the University of Edinburgh [4]
- GPGP, developed at the University of Massachusetts at Amherst, USA
- SIPE-2, developed at SRI International, Menlo Park, CA, USA
- UMCP and SHOP2, developed at University of Maryland, USA
- HTNPlan-P, developed at the University of Toronto, Canada

These systems and others like them use their own, system-specific representation languages for a textual representation of the Hierarchical Task Network and the domain model that underlies it. Some of these languages use one of the standard styles; for example the GPGP system uses the S-expression-style TÆMS [5] language (Task Analysis, Environmental Modeling and Simulation). Others use their own syntactic style, for example the Task Formalism (TF) of the O-Plan system.

In the present chapter we shall give particular attention to two of the above: The GPGP/TÆMS approach and the O-Plan system.

### 2.2.1 GPGP and TÆMS

Generalized Partial Global Planning (GPGP) is an approach to the design of HTN systems, and TÆMS is the representation system used in this approach. As such, TÆMS is both a modelling language that uses S-expressions as its syntactic style, and a design for the representation and use of Hierarchical Task Networks in the working software system.

We shall describe this approach by quotations. The following is the abstract of a key article describing GPGP: *Victor Lesser: Evolution of the GPGP/TÆMS domain-independent coordination framework. Proceedings of the first international joint conference on Autonomous agents and multiagent systems, 2002.*

Generalized Partial Global Planning (GPGP) and its associated TÆMS hierarchical task network representation were developed as a domain-independent framework for coordinating the real-time activities of small teams of cooperative agents working to achieve a set of high-level goals. GPGP's development was influenced by two factors: one was to generalize and make domain-independent the coordination techniques developed in the Partial Global Planning (PGP) framework (this also involved our understanding that coordination activities could be separated from local agent control if an appropriate bi-directional interface could be established between them); the other was based on viewing agent coordination in terms of coordinating a distributed search of a dynamically evolving goal tree. Underlying these two influences was a desire to construct a model that could be used to explain and motivate the reasons for coordination among agents based on a quantitative view of task/subproblem dependency. Coordination of behaviors among agents requires three things: specification (creating shared goals), planning (subdividing goals into subgoals/tasks, i.e., creating the substructure of the evolving goal tree) and scheduling (assigning tasks to

[4] http://www.aiai.ed.ac.uk/∼oplan/

[5] ftp://mas.cs.umass.edu/pub/TAEMSwhite.pdf

individual agents or groups of agents, creating shared plans and schedules and allocating resources). GPGP is primarily concerned with scheduling activities rather than the dynamic specification and planning of evolving activities (e.g., such as decomposing a high-level goal into a set of subgoals that if successfully achieved will solve the high-level goals).

In order to give some idea of the flavor of the TÆMS language, we cite the following example from the TÆMS White Paper, where it is described as being "probably the simplest complete task structure youll run across. It consists of a single root node, which well call a task group, which is the parent to a single child, which we'll call a method." (The semantics of these terms is defined later on in the document).

```
; The Task Group
(spec_task_group
   (label Root)
   (agent Agent_A)
   (subtasks Method_1)
   (qaf q_min) )

; The Method
(spec_method
   (label Method_1)
   (agent Agent_A)
   (supertasks Root)
   (outcomes
      (Outcome_1
         (density 1.0)
         (quality_distribution 10.0 1.0)
         (duration_distribution 6.0 1.0)
         (cost_distribution 5.0 1.0)
              )))
```

*The TÆMS White Paper further writes:*

TÆMS was designed as a modeling language for describing the task structures of agents. The acronym stands for Task Analysis, Environmental Modeling and Simulation. In this overview we will cover some of the high level ideas that underly this acronym and give TÆMS its capabilities. There is nothing in TÆMS that precludes it from being used outside of an agent or multi-agent system, but you will see in this document that most of the distinguishing characteristics of TÆMS are agent related, including its ability to represent capabilities of remote agents, and explicitly represent the interdependencies that exist between them and those of the local agent.

So what do we mean by "task structure"? In any sophisticated agent system, the designer will inevitably arrive at a point where the agent must reason about its potential actions in the context of its working environment. So, presented with a given situation, what should an agent do? What goals can and should it be trying to achieve? What actions are needed to achieve those goals? What are the implications of those actions, and of actions performed by remote agents, on the agents local state? There is a whole host of questions that an agent will need to be answered if it is to reason about its situation and act intelligently. Thus, the agent must have some representation of what its capabilities are. One such representation is called a task structure - something which describes the tasks the agent may perform. TÆMS improves upon conventional task structures by adding such features as quantitative action characterizations, explicit models of local and remote interactions and mechanisms to represent the wide range of ways a particular task can be achieved.

While the details associated with TÆMS can be daunting, there are in reality just a few simple concepts behind its structure and function. A TÆMS task structure is essentially an annotated task decomposition tree (actually a graph, but we will refer to is as a tree for simplicity here). The highest level nodes in the tree, called task groups, represent goals that an agent may try to achieve. Below a task group there will be a sequence of tasks and methods which describe how that task group may be performed. Tasks represent sub-goals, which can be further decomposed in the same manner. Methods, on the other hand, are terminal, and represent the primitive actions an agent can perform.

The structure above will work out to be a tree structure containing goals and sub-goals that can be achieved, along with the primitive methods needed to achieve them. Annotations on a task describe how its subtasks may be combined to satisfy it. Another form of annotation, called an interrelationship, describes how the execution of a method, or achievement of a goal, will affect other nodes in the structure. For instance, the execution of method A may enable the execution of method B. In other words, method B cannot be successfully performed before A is successfully completed. Several types of interrelationships exist to describe various types of situations.

*(End of quotations).*

### 2.2.2 The O-Plan System

The O-Plan system and approach were developed by Austin Tate and his group at the Artificial Intelligence Applications Institute (AIAI) at the University of Edinburgh. *The O-Plan webpage writes:*

The O-Plan (the Open PLANning Architecture) Project is exploring issues of coordinated command, planning and control.

The objective of the O-Plan Project is to develop an architecture within which different agents have command (task assignment), planning and execution monitoring roles.

Each agent has a structure which separates the following components:

- the representation of the processing capabilities of an agent;
- the computational facilities available to perform these capabilities;
- the constraint managers and commonly used support routines which are useful in the construction of command, planning and control systems;
- the decision making about what the agent should do next;
- the handling of communications between one agent and others.

The main contribution of the O-Plan research is to provide a complete vision of a more modular and flexible planning and control system incorporating AI methods.

We have demonstrated our approach on realistic problems related to military logistics. Our approach will have an impact in the following ways:

- it will allow for improved connectivity and consistency between command, planning and control;
- it will make plans open, inspectable, explainable and changeable;
- it will allow greater scope in Course of Action analysis and as such provide greater plan reliability.

Within the agent-based O-Plan architecture we have created specific agents to provide a domain-independent general planning and control framework with the ability to embed detailed knowledge of the domain. The system combines a number of techniques:

- A hierarchical planning system which can produce plans as partial orders on actions similar to the Edinburgh Nonlin planner.
- An agenda-based control architecture in which each control cycle can post pending tasks during plan generation. These pending tasks are then picked up from the agenda and processed by appropriate handlers (Knowledge Sources).
- The notion of a "plan state" which is the data structure containing the constraints on emerging plan, the "agenda" of further requirements, and the information used in building the plan.
- Constraint posting and least commitment on object variables.
- Temporal and resource constraint handling using incremental algorithms which are sensitively applied only when constraints alter.
- O-Plan is derived from the earlier Nonlin planner from which it takes and extends the ideas of Goal Structure, Question Answering (Modal Truth Criterion) and typed conditions.
- We have extended Nonlin's style of task description language Task Formalism (TF).

**Major Subsystems:**

- A **Task Assigner** specifies a task that is to be performed through some suitable interface.
- A **Planner** plans and (if requested) arranges to execute the plan to perform the task specified.
- The **Execution system** seeks to carry out the detailed tasks specified by the planner while working with a more detailed model of the execution environment.

*(End of quotation).*

## 2.3 Logic-Based Systems

*The contents of this section will be added later.*

### 2.3.1 IxTeT

The IxTeT system has been developed in the Robotics and A.I. group at the LAAS laboratory in Toulouse.

### 2.3.2 TALplanner and DyKnow

(TALplanner is a planning system; DyKnow is a middleware that provides support for executing plans generated by TALplanner).

### 2.3.3 The Golog System

*To be imported or written. Have not been able to find a good webpage for this yet.*

## 2.4 Procedural Systems

## 2.5 The PDDL Language

*Wikipedia writes:*

The Planning Domain Definition Language (PDDL) is an attempt to standardize planning domain and problem description languages. It was developed mainly to make the 1998/2000 International Planning Competitions possible. It was first developed by Drew McDermott in 1998 and later evolved with each International Planning Competitions.

Planning tasks specified in PDDL are separated into two files:

- A domain file for predicates and actions
- A problem file for objects, initial states and goal specifications

*End of citation*

PDDL is an S-expression style language. The original version of PDDL is specified in [6] and this is still a useful introduction. The PDDL+ version from 2006 is an extension of PDDL 2.1 and the article describing it [7] is recommended reading for learning about the current stage of development, since the additional changes for PDDL 3.1 (the most current version) are not very significant.

[6]http://www.informatik.uni-ulm.de/ki/Edu/Vorlesungen/GdKI/WS0203/pddl.pdf

[7]http://www.jair.org/media/2044/live-2044-2913-jair.pdf

# Chapter 3

# Languages and their Styles

The entire system state in the BDI model is a complex information object which therefore requires an expressive representation language, and the same applies for other models and approaches, such as SOAR, and the HTN approach. Some of its components may be relatively simple, and in particular the beliefbase is often organized merely as a set of literals, for example in the PRS system. However, plans and intentions are necessarily composite information objects that can best be thought of as formulas, or as simple programs of the kind that are often called scripts in programming terminology.

The choice of a representation language is therefore an important and necessary step in the development of the software architecture for an intelligent agent. There is a large variety of representation languages in the literature, but a few main lines are evident.

First of all, one can identify a small number of *syntactic styles,* that is, ways of writing formulas that are used in common between several representation languages. Lisp S-expressions is one syntactic style; XML notation is another syntactic style.

The underlying notion is that an intelligent agent system does not merely need one language for expressing its system state, it needs a number of related languages for different purposes. (The same is actually true for many other types of software as well, including for operating systems.) Rather than inventing an entirely new language for each purpose, it is convenient to define a general framework for how formulas are written, to implement general-purpose software for supporting that framework or style, and to have ways of specializing or adapting the framework and the software for each of the specific languages.

A syntactic style is just a set of conventions for writing formulas, and formulas can be used for many things, including for mathematical formulas, logic formulas that make statements about the real world, computer programs, chemical compounds, and many other things. The case of computer programs or scripts is of course of particular importance and many syntactic styles have one or more programming languages among their uses, but it is important to keep in mind that a style also has *other* uses besides for a programming language.

Styles also have a second important relation to programming languages, namely their use as a *data language,* that is, as a textual representation of data that can be processed by the programming language in question. Conventional programming languages such as C++ do not have any data language of importance, since the "print" operation is only defined for numbers and strings, but not for records, arrays or other composite data objects. However, languages like Python do provide a standardized way of writing and reading composite data, thereby having their own data language. In the particular case of Python the data language is ideosyncratic and used by Python only, but it is also possible to have a data language that belongs to a widely used syntactic style.

Languages like Lisp and Python are defined with a data language from the start. Other languages have obtained a data language as an add-on facility, and in these cases it is often referred to as a method of *serialization* of the data structures that are intrinsic to the language. This is the case for the Java language, in particular. *Have to check that this is really true.* The next section describes the most important syntactic styles that are used in representation languages for intelligent agents. The following chapter will describe major representation languages using these styles.

## 3.1 Major Language Styles

### 3.1.1 The S-Expression Style

The S-expression style has two characteristic features. First, there is a purely syntactic definition: S-expressions are expressions that are constructed recursively and where each level is surrounded by one kind of brackets; by convention the ordinary round parentheses are used as brackets. Such a sequence is called a *list.* The elementary components of S-expressions may be symbols, strings, or numbers. Strings are surrounded by double quote characters and may not contain double quotes inside them, except using an escape-character arrangement. The following are some examples of S-expressions.

```
A_symbol
"This is a string"
(This is "a list")
(This is (a list)(containing 2 sublists))
```

Secondly, the *S-expression style* also adopts a loose semantic convention, namely, that the first element in every list on every level shall be an operator that takes the remaining elements in the list as 'arguments' in one sense or another. This means that a mathematics-like notation that allows expressions such as, for example

```
(a + f(b + c) * c)
```

will not count as S-expression style, although from a purely syntactic viewpoint it qualifies as an S-expression. To be considered as S-expression style it would have to be written like, for example,

```
(+ a (* (f (+ b c)) c))
```

S-expressions were introduced in 1959 as the data language for the Lisp programming language. It was originally intended that Lisp should have a programming language that resembled conventional programming languages at the time, such as Algol, but it was also decided to define a way of representing Lisp programs as S-expressions, that is, to write programs in the data language. This was partly as an intermediate solution awaiting the implementation of a parser for the forthcoming, separate programming language, and partly in order to make it possible to analyze and manipulate Lisp programs as data. It soon became clear however that most users preferred to write Lisp programs directly in S-expression notation, so Lisp uses S-expressions *both* for expressing programs and for expressing data, which means that the two are easily exchangeable.

A number of other programming languages also use S-expressions in this way and are considered as descendants of Lisp, such as the Scheme language in particular.

There are a number of languages and formats in general computing, besides programming languages, that use S-expressions, including the Internet Message Access Protocol (IMAP) which is a standard for electronic mail exchange. Another example is the Document Style Semantics and Specification Language (DSSSL) which was later replaced by the Extensible Stylesheet Language (XSL). The TÆMS language that was mentioned in the previous chapter is an S-expression style language, and additional ones will be described later on in this compendium.

### 3.1.2 The Prolog Style

*Not clear whether this qualifies as a style, but keep the subheading here for the time being.*

### 3.1.3 The XML Style

The XML style originates from the Standard Generalized Markup Language (SGML) which was established in 1986 as an ISO-standard technology for defining generalized markup languages for documents. SGML was the basis for HTML which was introduced as the markup language for World-Wide Web pages in the early 1990's. After some time it was recognized that HTML was too restrictive but SGML was not an alternative since it was too complex, and XML was introduced as an intermediate solution.

One of the important aspects of XML was that it should be a representation both for text in natural language (the original purpose of SGML) and for structured data. It is in the latter sense that it is a language style for data languages. This dual use of the style has some advantages, namely, that it facilitates the integration of text and data, but it also has the important disadvantage that the XML representation of structured information becomes practically unreadable for the human reader except for reading small fragments. XML for structured information should therefore be considered as an exchange protocol that can be used for transmission of information between programs, but for human use it is more or less necessary to use viewing programs that display the information in a more user-friendly way.

*Write here about efforts to define scripts and a programming language using XML style.*

### 3.1.4 Knowledge Representation Expressions

Knowledge Representation Expressions are defined in our *KRF Overview* memo. They are similar to S-expressions in the sense that a KR expressions is a recursively formed, bracketed expression with symbols, strings, and numbers as its elements, but it differs from S-expressions by using several kinds of brackets, and not merely the round parentheses that are used by S-expressions. In particular, sets and sequences are represented by their standard notation in mathematics, as long as they are specified by enumerating their elements.

KR expressions also provide some additional structure within each bracketing level, in particular through a distinction between arguments and parameters in records, a *record* being an expression surrounded by square brackets. In this way KRE expressions are more readable than S-expressions, and of course much more readable than XML expressions.

There has not been a definition of a full programming language based on KR expressions, and the current software support for KRE is implemented in Lisp. KRE-based applications must therefore often be written in Lisp. However, the *Common Expression Language* (CEL) uses the KRE style and defines functional expressions (terms) in such a way that they can be used both for simple programming (scripts) and in logic formulas. It corresponds to the central parts of Lisp.

The need for seamless integration of text and structured data is supported in XML by using markup conventions that are reasonable for text but very clumsy for structured data. The KRE style has another solution for this problem, namely the use two closely related languages. The *Document Scripting Language* (DSL) uses KRE style without exceptions, so that it can be parsed using the standard KRE parser. It is useful for specifying texts that are included in structured data and texts that contain a large proportion of embedded data, which is for example often the case in scripts for dynamic web pages. The *Text Scripting Language* (TSL) uses somewhat different conventions which make it convenient for markup of source text, in the same way as Latex markup, but the TSL parser converts its input to the same datastructure representation as is used for all KR expressions, and in particular for DSL. This achieves the desired seamless integration of markup for text and representation of structured information, but retaining good legibility for both of these.

# Chapter 4

# Knowledge Representation Languages

One type of language that is needed in an intelligent agent architecture is for *representing beliefs* in the agent's belief base, and for transmission of facts and beliefs between agents. Languages of this kind are called *knowledge representation languages.* The most important characteristics of KR languages, in comparison with programming languages, is that the former refer to phenomena in the real world, such as persons, cars, colors, velocities, utterances, and so forth, whereas programming languages refer to numbers, strings, records and other similar structures in a computer. The semantics of a knowledge representation language must therefore specify how expressions in that language relate to things in the world.

The difference is not absolute, however, since things like 'numbers' and 'names' (represented as strings) must occur both in a programming language and in a representation language, and both have a need for the basic operations on these types. For example, a reasonable representation language should make it possible to express statements about the current air temperature, or the change in exchange rate between the Euro and the US Dollar from one day to the next. This overlap between the two types of languages may be exploited in either of two ways: by having a programming language and a representation language as two separate languages, which however coincide in some of their parts, or by having a single language that uses two different semantics, i.e., two different ways of evaluating expressions. The combination of the programming language Lisp and the representation language KIF is an example of the first approach; the KRE-based Common Expression Language is an example of the second approach. Two variants of evaluation are defined for CEL, namely *computational* and *representational* evaluation.

## 4.1 Extended Logic Representation Languages

### 4.1.1 Knowledge Interchange Format

The Knowledge Interchange Format (KIF) [1] is a knowledge representation language that uses the S-expression style, and that coincides with Lisp in some of its parts. As the name suggests it was originally designed for use in message exchange, in particular using the Knowledge Query and Manipulation Language (KQML) which also uses the S-expression style and which is described in a later chapter. However, KIF is also used for representing knowledgebases and ontologies, for example in the Suggested Upper Merged Ontology (SUMO) [2]

The following are some examples of expressions in KIF, all taken from the SUMO Ontology. First, a number of ground literals from the SUMO geography:

```
(instance MississippiRiver River)
(part MississippiRiver UnitedStates)
(documentation MississippiRiver EnglishLanguage
   "The major River in the &%UnitedStates.
    It runs almost the entire width of the UnitedStates, from &%Minnesota
    to the &%GulfOfMexico.")
(connected MississippiRiver GulfOfMexico)
```

Notice the possibility of using predicates with different arities, for example the ternary predicate `documentation.` Notice also the possibility of embedding entities within strings using the ampersand-percent prefix.

Next, a rule saying that Paris is the largest city in France in terms of population:

```
(=>
   (and
      (instance ?CITY City)
      (part ?CITY France))
   (lessThanOrEqualTo
     (CardinalityFn (ResidentFn ?CITY))
     (CardinalityFn (ResidentFn Paris))))
```

The function `Residentfn` maps a city to the set of its inhabitants, and is a clear example of how a representation language must refer to phenomena in the world, including those that can not have an explicit counterpart in the computer. It is of course not possible to have a representation in the computer for each one of the inhabitants of Paris, France, but this does not preclude the representation language for using an expression whose defined *value is* exactly the set of those persons.

Variables are preceded by the questionmark character in KIF, and are defined to be universally quantified in the absence of an explicit quantifier. Finally, a definition of the concept of a generalized union.

```
(deffunction generalized-union (?set) :=
```

[1] http://www.ksl.stanford.edu/knowledge-sharing/kif/
[2] http://www.ontologyportal.org/

```
  (if (and (set ?set)
           (forall (?s) (=> (member ?s ?set) (set ?s)))
      (setofall ?x (exists (?s) (and (member ?s ?set)
                                      (member ?x ?s) )))))
```

KIF can be understood as a development of Lisp that introduces additional concepts from set theory and predicate logic, besides switching from computational to predominantly representational evaluation. There is a function `listof` that works like the function `list` in Lisp; there is also a function `setof` that forms the set of those objects that occur as its arguments. The values of these functions are defined in the semantics of the language. The function `setofall` also forms a set of objects, but defined by a condition that is expressed in logic as in the example above.

### 4.1.2 The CycL Language

The CycL language is developed and used in the Cyc project [3] whose purpose is to build a very comprehensive, and therefore very large knowledgebase using an extended predicate-logic notation. The language is based on predicate logic, like KIF, and uses S-expression style. The following is a simple example of a statement in the Cyc language:

```
(#$implies
   (#$and
     (#$isa ?OBJ ?SUBSET)
     (#$genls ?SUBSET ?SUPERSET))
   (#$isa ?OBJ ?SUPERSET))
```

Here `isa` represents membership in a set or type and `genls` stands for 'generalisation' and represents the subset relation. This rule says that if $e$ is a member of $a$ and $a$ is a subset of $b$ then $e$ is a member of $b$. The `#$` character combination that precedes the logical symbols in this expression are optional, but widely used by convention.

### 4.1.3 The Prolog Language

*An account of Prolog as a representation language is maybe to be added here, or else combine this with ICL in Section 6.4.*

### 4.1.4 The Common Expression Language

*Discuss how it relates to the previous ones. Many points in common with KIF, but more legible through use of the variety of data types already on the style level: sets, sequences, records, and the associated use of multiple bracketing. The use of infix operations also increases readability considerably, bringing the language fairly close to standard notation especially in formal logic.*

[3]http://en.wikipedia.org/wiki/Cyc

## 4.2 XML-based Representation Languages

### 4.2.1 The Resource Definition Format

The Resource Definition Format (RDF) has been developed in the XML community and for representing information about objects in the Internet and the World-Wide Web, objects that are then called 'resources.' RDF is usually written using an XML style. It has been extended, by way of notation and of use, into the OWL language which will be described next.

### 4.2.2 The Web Ontology Language

OWL, the Web Ontology Language. is a language for characterizing entities, including both specific objects and classes of objects, as well as attributes of these entities and relationships between them. Particular attention is given to the subsumption relation because of its importance for representing ontologies.

Like RDF, OWL is usually expressed using an XML style notation, although in principle it is open to the use of other notations. The following is an example of a few entity descriptions in OWL, obtained from the OWL website [4]

```
<Region rdf:ID="SantaCruzMountainsRegion">
  <locatedIn rdf:resource="#CaliforniaRegion" />
</Region>

<Winery rdf:ID="SantaCruzMountainVineyard" />

<CabernetSauvignon
  rdf:ID="SantaCruzMountainVineyardCabernetSauvignon" >
  <locatedIn   rdf:resource="#SantaCruzMountainsRegion"/>
  <hasMaker    rdf:resource="#SantaCruzMountainVineyard" />
</CabernetSauvignon>
```

The information in these would be expressed as follows in KRE.

```
------------------------------------------------
-- SantaCruzMountainsRegion
[: type Region]
[: locatedIn CaliforniaRegion]

------------------------------------------------
-- SantaCruzMountainVineyard
[: type Winery]

------------------------------------------------
-- SantaCruzMountainVineyardCabernetSauvignon
[: type CabernetSauvignon]
[: locatedIn SantaCruzMountainsRegion]
[: hasMaker SantaCruzMountainVineyard]
```

[4]http://www.w3.org/TR/owl-features/

```
---------------------------------------------
```

## 4.3 Class Description Languages

The contents of this section are not to be confounded with *Description Logic* which will be discussed elsewhere in the present suite of compendiums.

### 4.3.1 The Language of the Knowledge Machine

The Knowledge Machine (KM) and its associated language have been developed by Bruce Porter, Peter Clark and their affiliates at the University of Texas at Austin, USA. The KM language uses S-expressions syntactically, but not exactly S-expression style since operators are arranged differently than in e.g. KIF and CycL. The following example is obtained from the KM website. [5] and shows how the class of buying actions is expressed in the KM language.

```
(Buy has (superclasses (Event)))                    ; Properties of the class
                                                    ; (‘owns’ properties)
(every Buy has                                      ; Properties of its members
  (buyer ((a Agent)))                               ’ (‘template’ properties)
  (object ((a Thing)))
  (seller ((a Agent)))
  (money ((the cost of (the object of Self))))
  (subevent1 ((a Give with
               (agent ((the buyer of Self)))
               (object ((the money of Self)))
               (rcpt ((the seller of Self))))))
  (subevent2 ((a Give with
               (agent ((the seller of Self)))
               (object ((the object of Self)))
               (rcpt ((the buyer of Self)))))))
```

This definition says that a buying action has attributes called buyer, seller, object and money, and two subevents namely one where the buyer gives money to the seller, and one where the seller gives the object to the buyer.

The KM language can therefore be characterized as a language for *expressing descriptions* of things and classes. The reader is recommended to study the very instructive script of an example session with the KM system [6]

## 4.4 API Approaches

Knowledge Representation Languages allow their user to write down information in formal, textual form and to work with that representation of the information, for example for having it reviewed by a panel of people. At the same time it is possible to input information in this form into intelligent agent systems as well as other software systems, where it can be used for a

[5]http://userweb.cs.utexas.edu/∼mfkb/km/
[6]http://userweb.cs.utexas.edu/users/mfkb/km/km-overview.script

variety of purposes. Knowledgebase contents can be expressed in the knowledge representation language, and can be shared between people, between agents, and between projects, and in this way the knowledge representation language becomes a medium of exchange.

However, there is also an alternative approach using an Application Programming Interface (API) where one defines a set of operations that can be implemented for each one of several different knowledgebases, each of which may use its own internal representation. This approach is useful if those knowledgebases are already in place and it is necessary to combine their use, or if the various knowledgebases serve different purposes which has made it necessary to organize them in different ways.

One important representative of the API approach is the Open Knowledge Base Connectivity (OKBC)

*Citation:* The OKBC is a protocol and an API for ontology repositories or object-relational databases. It is somewhat complementary to the Knowledge Interchange Format in the sense that it has been implemented as an API to KIF databases, together with several other representations. In this model, an ontology consists of a set of classes organized in a subsumption hierarchy to represent a domain's salient concepts, a set of slots associated to classes to describe their properties and relationships, and a set of instances of those classes - individual exemplars of the concepts that hold specific values for their properties.

Other OKBC implementations provide an API to LOOM which was a description-language database, and to Ontolingua which was a frame-structure knowledgebase. *More about frame structures needed here.* The Protégé software platform, developed at Stanford University, is an example of a widely used system that uses the OKBC knowledgebase interface, at the same time as it also uses the OWL representation language.

# Chapter 5

# Process-Controlling Agents

*This chapter has not been written yet. The following are some indications of its intended contents.*

Topic: how to organize systems where actions are executed in the real world using various kinds of sensors and/or controllers.

## 5.1 Three-Level Architectures

*The following is the suggested general approach for this section.* Simply speaking, the cognitive architeture may be used for the uppermost layer; the middle layer can be finite-state automaton, and the process layer may consist of control algorithms. However, there is no clear evidence in terms of actually implemented systems that this will work as intended.

The topic of this chapter is a very important one, but it will be only briefly covered in the present course.

## 5.2 Task Control Architecture

*The webpage of the original Task Control Architecture (TCA) [1] writes:* The Task Architecture (TCA) simplifies building task-level control systems for mobile robots. By "task-level", we mean the integration and coordination of perception, planning and real-time control to achieve a given set of goals (tasks). TCA provides a general control framework, and it is intended to be used to control a wide variety of robots. TCA provides a high-level, machine independent method for passing messages between distributed machines (including between Lisp and C processes). Although TCA has no built-in control functions for particular robots (such as path

[1]http://www.cs.cmu.edu/afs/cs/project/TCA/release/tca.orig.html

planning algorithms), it provides control functions, such as task decomposition, monitoring, and resource management, that are common to many mobile robot applications.

TCA can be thought of as a robot operating system — providing a shell for building specific robot control systems. Like any good operating system, the architecture provides communication with other tasks and the outside world, facilities for constructing new behaviors from more primitive ones, and means to control and schedule tasks and to handle the allocation of resources. At the same time, it imposes relatively few constraints on the overall control flow and data flow in any particular system. This enables TCA to be used for a wide variety of robots, tasks, and environments. *(End of quotation)*

In its later development [2] has been split into the *Inter-Process Communications facility,* IPC, and the *Task Description Language,* TDL, and its descendants. TDL is a superset of C++ that includes explicit syntax for task-level control capabilities.

## 5.3 TALplanner and DyKnow

*It's not clear at this point whether this topic should be in the present chapter or in chapter 2.*

## 5.4 The Subsumption Architecture

*Not yet written.*

[2]http://www.cs.cmu.edu/afs/cs/project/TCA/www/TCA-history.html

# Chapter 6

# Message-Passing Between Agents

## 6.1 Introduction

Many artificial intelligence systems, in particular in robotics, require the use of several concurrent processes. Examples include:

- Having one process for the movements and other physical behavior of an intelligent robot, and another process that defines its goal-directed behavior, planning, and the like
- Having a separate process for the agent's linguistic ability, when it is required to communicate in natural language
- Having a separate process for anticipatory simulation, as a way of predicting the immediate future of an agent
- Having a separate process for learning which can operate while the main system performs it normal tasks.

Powerful methods for the implementation of concurrent computational processes have been developed in the areas for programming languages and software engineering, using constructs such as software *semaphores.* However, methods such as these apply to conventional programs and they are not useful e.g. if one of the processes is organized according to the BDI model. In such cases the usual method is to let the processes communicate by passing *messages* to each other. (There are also other methods, such as the use of so-called *blackboards,* but these will not be covered in the present compendium). Given that one or more processes are organized using a software architecture for agents, it is common to consider each of the processes as an agent, so that the entire system is a *system of communicating agents.*

Notice that it is not necessary for all of the agents in such a system to be intelligent ones. It may well be that one of them is organized according to the BDI model, for example, and the others are fairly conventional programs. The point is however that then the BDI-model agent determines the character and the needs of the message-passing, and those agents that are more conventional programs have to be implemented so as to conform to those needs. In fact, in many cases it is questionable whether *any* of the agents deserves to be called ‘intelligent’ and it may be that only the combined system has that character.

Communication between agents may also occur for another reason, namely, if there are several systems that are more or less similar and that need to cooperate in order to achieve a common task. This is different from the situation we have described first, where the introduction of several agents is used in order to organize one single, coherent system. We use the term *individual* for a coherent system that consists of several agents performing different functions in that individual. Consequently we distinguish between message-passing *within* an individual, i.e. between the agents that make up the individual, and message-passing *between* individuals.

There are cases where it is a matter of definition what is an individual and what subsystems are merely parts of a single individual. This is just like, in the realm of zoology, one may view ants as separate individuals, but for some purposes one may also view the entire ant-heap as one individual although its different ‘parts’ can move freely relative to each other. However, in most cases this is not an issue, and the individuals of the overall system can be identified easily.

A particularly interesting case, especially from a philosophical point of view, concerns individuals that have a system state (in the sense of Section 1.1) that persists over a relatively long period of time, such as months or even years, and that is able to evolve during that time by the accumulation of memory, such as records of past events, and by the successive learning of new capabilities. Such individuals are called *persistent individuals.*

## 6.2 Searle’s Speech Act Theory

The major approach to message-passing in A.I. systems has been inspired by Searle’s *theory of speech acts.* Searle’s theory is rich in content and has developed over time, and it is not possible to make full justice to it here. The following should be taken as merely an indication of some major points in the theory. Some segments of text have been obtained from the Wikipedia article on speech act theory.

Searle uses the well established distinction between the ’illocutionary force’ and the ’propositional content’ of an utterance. He does not precisely define the former as such, but rather introduces several possible illocutionary forces by example. According to this terminology, the sentences

```
Sam smokes habitually.
Does Sam smoke habitually?
Sam, smoke habitually!
Would that Sam smoked habitually!
```

in their plain reading indicate the same propositional content (Sam smoking habitually) but differ in the illocutionary force indicated (a statement, a question, a command, and an expression of desire, respectively).

The analysis of utterances in terms of illocutionary force and propositional content is relevant for the analysis of dialogs, that is, for addressing the problem that was illustrated by the "Bath Assistance Scenario" in our introduction to the goals of A.I. research. A simple assertion phrase, such as "I can not do that myself" may have different illocutionary force according to context, where being a statement is one alternative and being a request for assistance (that is, a kind of command) is another one.

From the point of view of speech act theory it is natural to consider the illocutionary aspect of the utterance as the major one, and to view it as a particular kind of action, called an *illocutionary act.* The propositional content of the utterance is then seen as a kind of argument or parameter to the illocutionary act.

One significant property of illocutionary acts, according to Searle, is that they are characterised by their having *conditions of satisfaction* and a *direction of fit.* For example, the statement "John bought two candy bars" is satisfied if and only if it is true, i.e. John did buy two candy bars. By contrast, the command "John, buy two candy bars" is satisfied if and only if John carries out the action of purchasing two candy bars. Searle refers to the first as having the word-to-world direction of fit, since the words are supposed to accurately represent the world, and the second as having the world-to-word direction of fit, since the world is supposed to change to match the words. There is also the double direction of fit, in which the relationship goes both ways, and the null or zero direction of fit, in which it goes neither way because the propositional content is presupposed, as in "I'm sorry I ate John's candy bars."

When a listener in a dialog receives an utterance, the propositional content is often quite explicit, but the illocutionary content is sometimes implicit and the listener has the task of identifying it. This may be done by considering a range of possible illocutionary content and by checking, for each of them, whether its conditions of satisfaction are applicable and whether they make sense.

*Need a good example here.* Searle has also worked extensively on other topics, including the topic of intensionality, but his work on illocutionary acts is what is particularly relevant for the design of message-passing systems. More specifically, it is the concepts of satisfaction conditions and direction of fit that are of concrete interest, whereas the mechanisms for recognizing implicit illocutionary content are not so applicable to communication between software agents, at least with the present level of technology.

## 6.3 Message-Passing Infrastructure

We proceed now to the description of languages for the message-passing infrastructure that is needed in systems of communicating intelligent agents.

### 6.3.1 The KQML Language

The KQML language was developed around 1990 and was adopted in particular in the DARPA AI community as a standard for the representation of messages between agents. KQML stands for "Knowledge Query and Manipulation Language," where a 'query' is to be interpreted as an information request from one agent to another, and not as e.g. a database query. It uses the syntactic style of S-expressions and is organized as a set of *communicative acts;* this term replaces the term “speech acts” since software agents do not use speech for communication between them. These communicative acts are formally defined using their satisfaction conditions, along the lines of Searle’s speech act theory.

The KQML language was the basis for a standardization effort in the FIPA (The Foundation for Intelligent Physical Agents; existed from 1996 to 2005) which was a body for developing and proposing computer software standards for heterogeneous and interacting agents and agent-based systems. The FIPA “standard” called the Agent Communication Language (FIPA-ACL) is the most widely used one of FIPA’s proposed standards. After the dissolution of FIPA its work continues in an IEEE standards committee.

### 6.3.2 The Agent Communication Language

The specification of ACL consists of a set of communicative acts and the description of their *pragmatics,* that is, the intended effects on the system state of the sender and receiver agents. As an adaptation to the technical context, communicative acts are also called *message types.* The pragmatics of a message type is similar to the satisfaction condition but it is more limited since it does not take changes in the physical world into account.

The satisfaction condition of a communicative act is described with both a narrative form and a formal semantics in a specification language that is based on modal logic.

#### ACL Message Structure

The following is a simple example of a message in the ACL language in its transport form, that is, a form that can be used when the message is transmitted from one agent to another.

```
(inform
   :sender agent1
   :receiver hpl-auction-server
   :content (price (bid good02) 150)
   :in-reply-to round-4
   :reply-with bid04
   :language sl
   :ontology hpl-auction)
```

In their transport form, messages are represented as S-expressions. The first element of the message is a word which identifies the message type or communicative act being communicated. There then follows a sequence of message parameters, introduced by parameter keywords beginning with a colon character. No space appears between the colon and the parameter keyword. One of the parameters contains the content of the message, similar to the propositional content in Searle’s terms. The contents is encoded as an expression in some formalism, and the choice of formalism is indicated by the `language` parameter. Other parameters help the message transport service to deliver the message correctly (e.g. sender and receiver), help the receiver to interpret the meaning of the message (e.g. language and ontology), or help the receiver to respond co-operatively (e.g. reply-with, reply-by).

The ACL language can be specified using two separate lists: a list of recommended communicative acts, and a list of parameter keywords. Many of the parameters are used by several of the communicative acts, so it is not necessary to specify a separate list of parameters for each act.

## Communicative Acts

The following is the recommended set of communicative acts according to the published standards document of FIPA,

`http://www.fipa.org/specs/fipa00037/SC00037J.html#_Toc26729697`

*I think underscore characters are missing here.*

- ***Accept Proposal***
- ***Agree***
- ***Cancel***
- ***Call for Proposal***
- ***Confirm***
- ***Disconfirm***
- ***Failure***
- ***Inform*** Content is a proposition. Sender holds it to be true, wishes the receiving agent to also hold it to be true, and does not already believe that this is the case.
- ***Inform if***
- ***Inform Ref*** Content should consist of two arguments both of which are terms. The first term is not to be evaluated by the receiving agent; the second term is to be evaluated. The purpose of the act is to inform the receiving agent that the value of the first term equals the value of the second term. The sending agent must assume that the receiving agent is able to evaluate the second term but not the first term.
- ***Not Understood*** Sender informs receiver that it observed an action by receiver but did not understand it. Special case: received message from it but did not understand the message.
- ***Propagate*** Content specifies embedded message to be forwarded, and a set of agents to forward it to.
- ***Propose*** Propose receiver to perform action given in contents, provided that condition given in contents is true.
- ***Proxy***
- ***Query If***
- ***Refuse*** Response to request act when not willing or able to perform the act.
- ***Reject Proposal***

- ***Request*** Request receiver to perform a particular action.
- ***Request When***
- ***Request Whenever***
- ***Subscribe***

## Parameters in ACL Messages

The following are the available message parameters in the ACL standard. Additional parameters may be added in specific applications.

`:sender` Denotes the identity of the sender of the message, i.e. the name of the agent of the communicative act.

`:receiver` Denotes the identity of the intended recipient of the message. Note that the recipient may be a single agent name, or a tuple of agent names. This corresponds to the action of multicasting the message. Pragmatically, the semantics of this multicast is that the message is sent to each agent named in the tuple, and that the sender intends each of them to be recipient of the CA encoded in the message. For example, if an agent performs an inform act with a tuple of three agents as receiver, it denotes that the sender intends each of these agent to come to believe the content of the message.

`:content` Denotes the content of the message; equivalently denotes the object of the action.

`:reply-with` Introduces an expression which will be used by the agent responding to this message to identify the original message. Can be used to follow a conversation thread in a situation where multiple dialogues occur simultaneously.

`:in-reply-to` Denotes an expression that references an earlier action to which this message is a reply.

For example, if agent $i$ sends to agent $j$ a message which contains `:reply-with` ***query1*** then agent $j$ will respond with a message containing `:in-reply-to` ***query1.***

`:envelope` Denotes an expression that provides useful information about the message as seen by the message transport service. The content of this parameter is not defined in the specification, but may include time sent, time received, route, etc. The structure of the envelope is a list of keyword value pairs, each of which denotes some aspect of the message service.

`:language` Denotes the encoding scheme of the content of the action.

`:ontology` Denotes the ontology which is used to give a meaning to the symbols in the content expression.

`:reply-by` Denotes a time and/or date expression which indicates a guideline on the latest time by which the sending agent would like a reply.

`:protocol` Introduces an identifier which denotes the protocol which the sending agent is employing. The protocol serves to give additional context for the interpretation of the message.

`:conversation-id` Introduces an expression which is used to identify an ongoing sequence of communicative acts which together form a conversation. A conversation may be used by an agent to manage its communication strategies and activities. In addition the conversation may provide additional context for the interpretation of the meaning of a message.

Notice that several of these parameters require the corresponding value to be a composite data expression. This applies for example for `:envelope` and `:reply-by` and maybe also for `:conversation-id,` besides of course for `:content.` The syntax for these values will therefore depend on the conventions in each application. For example, if the Common Expression Language is used as the content language, then it is natural to express these other values as KR expressions as well, for example representing the value of the `:envelope` parameter as a KRE mapping.

### The Message Content Component

*The contents of this section have been obtained from the FIPA-ACL webpage.* The content of a message refers to whatever the communicative act applies to. If, in general terms, the communicative act is considered as a sentence, the content is the grammatical object of the sentence. In general, the content can be encoded in any language, and that language will be denoted by the `:language` parameter. The ACL language specification is therefore open with respect to content language, but it defines a precise requirement on the content language as follows (Requirement 6 in the language specification):

*In general, a content language must be able to express propositions, objects and actions. No other properties are required, though any given content language may be much more expressive than this. More specifically, the content of a message must express the data type of the action: propositions for inform, actions for request, etc.*

A proposition states that some sentence in a language is true or false. An object, in this context, is a construct which represents an identifiable "thing" (which may be abstract or concrete) in the domain of discourse. Object in this context does not necessarily refer to the specialised programming constructs that appear in object-oriented languages like `C++` and Java. An action is a construct that the agent will interpret as being an activity which can be carried out by some agent.

**Our comment:** These requirements in the FIPA-ACL specification mean that a content language for ACL must be a *knowledge representation language* in the sense that was introduced in Chapter 4, and that candidate content languages include KIF, the Cyc language, and the Common Expression Language. Notice that KIF was developed together with KQML and that ACL is a direct successor of the latter.

In general, an action is supposed to produce an effect in the environment and not a result that is communicated to another agent, but there are exceptions, in particular the `iota` operator in KIF which is expressed as `(iota <variable> <term>).` This operator introduces a scope for the given expression (which denotes a term), in which the given identifier, which would otherwise be free, is defined. An expression containing a free variable is not a well-formed SL expression. The expression `(iota x (P x)` may be read as “the x such that P [is true] of x.” The iota operator is a constructor for terms which denote objects in the domain of discourse.

### The JADE Implementation of FIPA-ACL

The FIPA-ACL standard has been implemented i.a. as the Java Agent DEvelopment Framework (JADE). The following is an excerpt from the JADE website.

*The JADE Agent Platform complies with FIPA specifications and includes all those mandatory components that manage the platform, that is the ACC, the AMS, and the DF. All agent communication is performed through message passing, where FIPA ACL is the language to represent messages. The agent platform can be distributed on several hosts. Only one Java application, and therefore only one Java Virtual Machine (JVM), is executed on each host. Each JVM is basically a container of agents that provides a complete run time environment for agent execution and allows several agents to concurrently execute on the same host.*

*The communication architecture offers flexible and efficient messaging, where JADE creates and manages a queue of incoming ACL messages, private to each agent; agents can access their queue via a combination of several modes: blocking, polling, timeout and pattern matching based. The full FIPA communication model has been implemented and its components have been clearly distincted and fully integrated: interaction protocols, envelope, ACL, content languages, encoding schemes, ontologies and, finally, transport protocols. The transport mechanism, in particular, is like a chameleon because it adapts to each situation, by transparently choosing the best available protocol. Java RMI, event-notification, HTTP, and IIOP are currently used, but more protocols can be easily added via the MTP and IMTP JADE interfaces. Most of the interaction protocols defined by FIPA are already available and can be instantiated after defining the application-dependent behaviour of each state of the protocol. SL and agent management ontology have been implemented already, as well as the support for user-defined content languages and ontologies that can be implemented, registered with agents, and automatically used by the framework.*

## 6.4 The Interagent Communication Language and OAA

The *Interagent Communication Language* (ICL) is the language used by the *Open Agent Architecture* (OAA). See websites at

```
http://www.ai.sri.com/~cheyer/papers/aai/node12.html
http://www.ai.sri.com/software/OAA
```

ICL is in some ways more advanced than ACL and it also incorporates important aspects of Prolog. We shall not use ICL material in the sequel but we include an overview of OAA and ICL here in order to broaden the perspective on agent communication systems.

*The OAA website says:* The OAA is a framework for building distributed communities of agents, where agent is defined as any software process that meets the conventions of the OAA society. An agent satisfies this requirement by registering the services it can provide in an acceptable form, by being able to speak the Interagent Communication Language (ICL), and by sharing functionality common to all OAA agents, such as the ability to install triggers, manage data in certain ways. *(Copied from the OAA website.)*

*The ICL website says:* OAA's Interagent Communication Language (ICL) is the interface, communication, and task coordination language shared by all agents, regardless of what platform they run on or what computer language they are programmed in. ICL is used by an agent to task itself or some subset of the agent community, either using explicit control or, more frequently, in an underspecified, loosely constrained manner. OAA agents employ ICL to perform queries, execute actions, exchange information, set triggers, and manipulate data in the agent community.

One of the fundamental program elements expressed in ICL is the *event.* The activities of every agent, as well as communications between agents, are structured around the transmission and handling of events. In communications, events serve as messages between agents; in regulating the activities of individual agents, they may be thought of as goals to be satisfied.

Each event has a *type,* a *set of parameters,* and *content.* For example, the agent library procedure `oaa_Solve` can be used by an agent to request services of other agents. A call to `oaa_Solve,` within the code of agent A, results in an event having the form

```
ev_post_solve(Goal, Params)
```

going from A to the facilitator, where `ev_post_solve` is the type, Goal is the content, and Params is a list of parameters. The allowable content and parameters vary according to the type of the event. [*The OAA facilitator is an intermediating process that can broadcast a service request to available agents and return the positive answers to the initiating agent.*]

The ICL includes a layer of conversational protocol, similar in spirit to that provided by KQML, and a content layer, analogous to that provided by KIF. The conversational layer of ICL is defined by the event types, together with the parameter lists associated with certain of these event types. The content layer consists of the specific goals, triggers, and data elements that may be embedded within various events.

The conversational protocol is specified using an orthogonal, parameterized approach. That is, the conversational aspects of each element of an interagent conversation are represented by a selection of an event type, in combination with a selection of values for an orthogonal set of parameters. This approach offers greater expressiveness than an approach based solely on a fixed selection of speech acts, such as embodied in KQML. For example, in KQML, a request to satisfy a query can employ either of the performatives `ask_all` or `ask_one.` In ICL, on the other hand, this type of request is expressed by the event type `ev_post_solve,` together with the `solution_limit(N)` parameter - where N can be any positive integer. (A request for all solutions is indicated by the omission of the `solution_limit` parameter.) The request can also be accompanied by other parameters, which combine to further refine its semantics.

In KQML, then, this example forces one to choose between two possible conversational options, neither of which may be precisely what is desired. In either case, the performative chosen is a single value that must capture the entire conversational characterization of the communication. This requirement raises a difficult challenge for the language designer, to select a set of performatives that provides the desired functionality without becoming unmanageably large. Consequently, the debate over the right set of performatives has consumed much discussion within the KQML community.

The content layer of the ICL has been designed as an extension of the Prolog programming language, to take advantage of unification and other features of Prolog. OAA's agent libraries (especially the non-Prolog versions) provide support for constructing, parsing, and manipulating ICL expressions

*Check: is it appropriate to consider Prolog vs ICL content layer language as a language couple that is analogous to Lisp vs KIF ?*

## 6.5 Leonardo Message-Passing Infrastructure

*This section is to be extended.* The Leonardo system uses a system for message-passing between agents that largely resembles the structure described above, although only a subset of the ACL-FIPA communication acts have been implemented at this point. The lower levels of the implementation uses the HTTP protocol for the transmission of messages, and complements it with a small amount of additional conventions and code.

There are two major types of messages, namely *requests* and *responses,* both of which are essentially records although some additional features are added and their exact format is modified in order to fit the HTTP-based message-passing infrastructure.

## 6.6 The CORBA Architecture

*The Wikipedia article on CORBA [1] says:* The *Common Object Request Broker Architecture* (CORBA) is an architecture that enables separate pieces of software written in different languages and running on different computers to work together as a single application or set of services. More specifically, CORBA is a mechanism in software for normalizing the method-call semantics between application objects that reside either in the same address space (application) or remote address space (same host, or remote host on a network). Version 1.0 was released in October 1991.

Implementations exist for Lisp, Ruby, Smalltalk, Java, COBOL, PL/I and Python. There are also non-standard mappings for Perl, Visual Basic, Erlang, and Tcl implemented by object request brokers (ORBs) written for those languages.

The CORBA specification dictates that there shall be an ORB through which the application interacts with other objects. In practice, the application simply initializes the ORB, and accesses an internal Object Adapter,

[1]http://en.wikipedia.org/wiki/CORBA

which maintains things like reference counting, object (and reference) instantiation policies, and object lifetime policies. The Object Adapter is used to register instances of the generated code classes. Generated code classes are the result of compiling the user IDL code, which translates the high-level interface definition into an OS- and language-specific class base for use by the user application. This step is necessary in order to enforce CORBA semantics and provide a clean user process for interfacing with the CORBA infrastructure. *End of quotation.*

This means that CORBA differs in important ways from the communication languages that have been described in earlier sections. Unlike them, CORBA is “mere architecture” and it does not define a communication language. The IDL that is mentioned above is the Interface Definition Language. The characteristic properties of CORBA help performance, making it a very appropriate choice for many robotic and process-control applications, but the same properties fail to meet all the needs of cognitive-level applications. Uses of CORBA in intelligent robotics must therefore be complemented with additional systems or layers that support the cognitive level of the overall system.

# Chapter 7

# Specialized Top-Level Platforms

*This chapter is intended to cover tasks that do not fit directly into the generalized robotic paradigm of earlier chapters, and where models such as the BDI model or the HTN model do not have much relevance either.*

# Chapter 8

# Ontologies

## 8.1 Content and Purpose of Ontologies

*The Wikipedia article on ontology in computer science [[1]] says:*

In computer science and information science, an *ontology* is a formal representation of the knowledge by a set of concepts within a domain and the relationships between those concepts. It is used to reason about the properties of that domain, and may be used to describe the domain.

In theory, an ontology is a "formal, explicit specification of a shared conceptualisation." An ontology provides a shared vocabulary, which can be used to model a domain that is, the type of objects and/or concepts that exist, and their properties and relations.

Ontologies are used in artificial intelligence, the Semantic Web, systems engineering, software engineering, biomedical informatics, library science, enterprise bookmarking, and information architecture as a form of knowledge representation about the world or some part of it. The creation of domain ontologies is also fundamental to the definition and use of an enterprise architecture framework.

Contemporary ontologies share many structural similarities, regardless of the language in which they are expressed. As mentioned above, most ontologies describe individuals (instances), classes (concepts), attributes, and relations. Common components of ontologies include:

- Individuals: instances or objects (the basic or "ground level" objects)
- Classes: sets, collections, concepts, classes in programming, types of objects, or kinds of things.
- Attributes: aspects, properties, features, characteristics, or parameters that objects (and classes) can have
- Relations: ways in which classes and individuals can be related to one another

[1] http://en.wikipedia.org/wiki/Ontology_(information_science)

- Function terms: complex structures formed from certain relations that can be used in place of an individual term in a statement
- Restrictions: formally stated descriptions of what must be true in order for some assertion to be accepted as input
- Rules: statements in the form of an if-then (antecedent-consequent) sentence that describe the logical inferences that can be drawn from an assertion in a particular form
- Axioms: assertions (including rules) in a logical form that together comprise the overall theory that the ontology describes in its domain of application. This definition differs from that of "axioms" in generative grammar and formal logic. In those disciplines, axioms include only statements asserted as a priori knowledge. As used here, "axioms" also include the theory derived from axiomatic statements.
- Events: the changing of attributes or relations.

Ontologies are commonly encoded using ontology languages.

### 8.1.1 Formal vs Lightweight Ontologies

*Citation from [2]*

A variety of ontologies form a continuum from lightweight, rather informal, to heavyweight, and formal ontologies. The lightweight ontology approach and the formal ontology approach are often used differently and have different strengths and weaknesses. Lightweight ontologies usually are taxonomies, which consist of a set of concepts (i.e., terms, or atomic types) and hierarchical relationships among the concepts. It is relatively easy to construct a lightweight ontology. To use a lightweight ontology for interoperability purposes, all parties need to agree on the exact meaning of the concepts. Reaching such agreements can be difficult. The lightweight ontology and the agreements together form a standard that all parties uniformly adopt and implement. That is, a lightweight ontology is often used to support strict data standardization. In contrast, a formal ontology uses axioms to explicitly represent subtleties and has inference capabilities. It can support data standardization in a different way, that is, the agreements are explicitly specified in the ontology. More often, a formal ontology is used to allow for data heterogeneity and to support interoperability, in which case the different interpretations and representations of data are explicitly captured in the ontology. In either case, a formal ontology disambiguates all concepts involved. Once created, a formal ontology can be relatively easy to use. But it often takes tremendous effort to create a formal ontology due to the level of detail and complexity required.

To summarize, lightweight ontologies are often used as data standards; as artifacts, they are simple, and thus easy to create, but difficult to use. Formal ontologies are often used to support interoperability of heterogeneous data sources and receivers; as artifacts, they are complex and difficult to create, but easy to use. Either approach has its weakness that limits its effectiveness.

[2]http://web.mit.edu/smadnick/www/wp/2006-06.pdf

*(The Wikipedia article on lightweight ontologies is not recommended).*

### Domain ontologies and upper ontologies

The following distinction is mostly applicable for formal ontologies; lightweight ontologies typically do not use it or need it.

A domain ontology (or domain-specific ontology) models a specific domain, or part of the world. It represents the particular meanings of terms as they apply to that domain. For example the word card has many different meanings. An ontology about the domain of poker would model the "playing card" meaning of the word, while an ontology about the domain of computer hardware would model the "punched card" and "video card" meanings.

An upper ontology (or foundation ontology) is a model of the common objects that are generally applicable across a wide range of domain ontologies. It contains a core glossary in whose terms objects in a set of domains can be described. There are several standardized upper ontologies available for use, including Dublin Core, GFO, OpenCyc/ResearchCyc, SUMO, and DOLCE. WordNet, while considered an upper ontology by some, is not strictly an ontology. However, it has been employed as a linguistic tool for learning domain ontologies.

The Gellish ontology is an example of a combination of an upper and a domain ontology.

Since domain ontologies represent concepts in very specific and often eclectic ways, they are often incompatible. As systems that rely on domain ontologies expand, they often need to merge domain ontologies into a more general representation. This presents a challenge to the ontology designer. Different ontologies in the same domain can also arise due to different perceptions of the domain based on cultural background, education, ideology, or because a different representation language was chosen.

## 8.2 Ontology Languages

Most of the representation languages that were described in Chapter 4 are used for representing ontologies, in particular KIF, CycL and OWL. The following are some additional languages that have been developed specifically for representing ontologies (quotation from the Wikipedia article on ontologies [3] )

- The *Common Algebraic Specification Language* is a general logic-based specification language developed within the IFIP (International Federation of Information Processing) working group 1.3 "Foundations of System Specifications" and functions as a de facto standard in the area of software specifications. It is now being applied to ontology specifications in order to provide modularity and structuring mechanisms.
- *Common logic* is ISO standard 24707, a specification for a family of ontology languages that can be accurately translated into each other.

[3]http://en.wikipedia.org/wiki/Ontology_(information_science)

- *DOGMA* (Developing Ontology-Grounded Methods and Applications) - additional information not available.
- The *Gellish language* includes rules for its own extension and thus integrates an ontology with an ontology language.
- The *Integrated Definition for Ontology Description Capture Method* (IDEF5) is a software engineering method to develop and maintain usable, accurate, domain ontologies.
- The *Rule Interchange Format* (RIF) and *F-Logic* combine ontologies and rules.
- The *Semantic Application Design Language* (SADL) captures a subset of the expressiveness of OWL, using an English-like language entered via an Eclipse Plug-in.
- OBO, a language used for biological and biomedical ontologies.

### 8.2.1 The RIF Language

*Wikipedia says:*
RIF is part of the infrastructure for the semantic web, along with (principally) RDF and OWL. Although originally envisioned by many as a "rules layer" for the semantic web, in reality the design of RIF is based on the observation that there are many "rules languages" in existence, and what is needed is to exchange rules between them. *End of citation.*

In practice RIF is therefore a family of languages, rather than a single language, and in terms of the present compendium one may consider RIF as a language style and its various so-called dialects as languages using that style. The following are simple examples from some of these ‘dialects’.

#### The Production Rules Dialect

The Production Rules Dialect (PRD) can be used to model production rules. Notably features in PRD include negation and retraction of facts (thus, PRD is not monotonic). PRD rules are order dependent, hence conflict resolution strategies are needed when multiple rules can be fired. The PRD specification defines one such resolution strategy based on forward-chaining reasoning.

```
Prefix(ex <http://example.com/2008/prd1#>)
(* ex:rule_1 *)
Forall ?customer ?purchasesYTD (
 If   And( ?customer#ex:Customer
           ?customer[ex:purchasesYTD->?purchasesYTD]
           External(pred:numeric-greater-than(?purchasesYTD 5000)) )
 Then Do( Modify(?customer[ex:status->"Gold"]) ) )
```

#### The Uncertainty Rule Dialect

The Uncertainty Rule Dialect (URD) supports a direct representation of uncertain knowledge, as in the following example.

```
Document(
  Import (<http://example.org/fuzzy/membershipfunction >)
  Group
  (
    Forall ?x ?y(
        cheapFlight(?x ?y) :- affordableFlight(?x ?y)
    )  / 0.4
    Forall ?x ?y(affordableFlight(?x ?y))  / left_shoulder0k4k1k3k(?y)
  )    )
```

It is debatable whether these RIF languages should best be described as knowledge representation languages, specialized ontology languages or high-level programming languages.

### 8.2.2 Common Logic

*Wikipedia writes:*
Common logic (CL) is a framework for a family of logic languages, based on first-order logic, intended to facilitate the exchange and transmission of knowledge in computer-based systems.

The CL definition permits and encourages the development of a variety of different syntactic forms, called "dialects." A dialect may use any desired syntax, but it must be possible to demonstrate precisely how the concrete syntax of a dialect conforms to the abstract CL semantics, which are based on a model theoretic interpretation. Each dialect may be then treated as a formal language. Once syntactic conformance is established, a dialect gets the CL semantics for free, as they are specified relative to the abstract syntax only, and hence are inherited by any conformant dialect. In addition, all CL dialects are equivalent (i.e., can be mechanically translated to each other), although some may be more expressive than others.

The standard includes specifications for three dialects, the Common Logic Interchange Format (CLIF), the Conceptual Graph Interchange Format (CGIF), and an XML-based notation for Common Logic (XCL). The semantics of these dialects are defined by their translation to the abstract syntax and semantics of Common Logic. Many other logic-based languages could also be defined as subsets of CL by means of similar translations; among them are the RDF and OWL languages, which have been defined by the W3C (World-Wide Web Consortium). *End of citation.*

The following is a simple example of these formats:

```
CLIF:  (exists (x y) (and (Red x) (not (Ball x)) (On x y)
                          (not (and (Table y) (not (Blue y)))) ))
CGIF:  ~[[*x] [*y] (Red ?x) ~[(Ball ?x)] (On ?x ?y)
                    ~[(Table ?y) ~[(Blue ?y)]]]
```

This shows how CLIF uses S-expression style and CGIF uses its own special format for representing logic formulas using the standard (Latin-1) computer character set. The peculiar conventions of CGIF include the use of square brackets for enclosing a conjunction (and-expression) and a universal quantifier, and the use of the asterisk for marking the quantification of a variable.

Notice that the concept of dialect in Common Logic is quite different from dialects in RIF. RIF dialects are languages for different purposes that use a more or less common style, whereas the point with Common Logic dialects is to allow different styles for the same, or at least overlapping semantic content.

### 8.2.3 The Gellish Language

*The Wikipedia article about Gellish [4] writes:*
Gellish is a controlled natural language, also called a formal language, in which information and knowledge can be expressed in such a way that it is computer-interpretable, as well as system-independent. Gellish is a structured subset of natural language that is suitable for information modelling and knowledge representation and as a successor of electronic data interchange. From a data modeling perspective, it is a generic conceptual data model that also includes domain-specific knowledge and semantics. Therefore, it can also be called a semantic data model. The accompanying Gellish modelling method thus belongs to the family of semantic modelling methods.

The data model in Gellish is based on binary relations between entities, similar to the model in OWL.

Etymologically speaking, "Gellish" is originally derived from "Generic Engineering Language." However, it is further developed into a language that is also applicable outside the engineering discipline.

### 8.2.4 The SADL Language

A simple example of the use of SADL, from its webpage:

```
shapes-top.sadl
   uri "http://ctp.geae.ge.com/iws/shapes_top".

   Shape is a top-level class.
   area describes Shape has values of type float.

shapes-specific.sadl
   uri "http://ctp.geae.ge.com/iws/shapes_specific".

   import "file://shapes-top.sadl" as shapes-top.

   Circle is a type of Shape.
   radius describes Circle has values of type float.

   Rectangle is a type of Shape.
   height describes Rectangle has values of type float.
   width describes Rectangle has values of type float.
```

Reasoning over a set of SADL documents takes two basic forms. *Validation* of a model involves checking the model for contradictions or inconsistencies. *Rule processing* involves examining the rules in the model in light of

[4]http://en.wikipedia.org/wiki/Gellish

the current instance data to see if any of the rules can "fire" to infer additional information. Two reasoners are integrated with the SADL Integrated Development Environment (SADL-IDE).

A systematic transformation from SADL to OWL has been defined.

### 8.2.5 The IDEF5 Method and Language

The digit '5' in the acronym IDEF5 is not a version generation number, but represents the fact that there is an IDEF family of modelling languages that serve different and complementary purposes, and IDEF5 is the particular language used for ontologies in this family. This family of languages was developed by the U.S. Air Force in the early 1990's, and is presently maintained and used by a commercial company, Knowledge Based Systems, Inc.

The IDEF5 method has three main components: A graphical language to support conceptual ontology analysis, a structured text language for detailed ontology characterization, and a systematic procedure that provides guidelines for effective ontology capture. The graphical language appears to be the primary representation.

## 8.3 Published Formal Ontologies

A large number of proposed formal ontologies have been published, both general-purpose ones and specialized ontologies for different disciplines or areas of knowledge or application. The following are some of the more important general-purpose ontologies.

- The Cyc ontology, [5]
- The Suggested Upper Merged Ontology, SUMO, [6]
- The Generalized Upper Model, GUM
- The Descriptive Ontology for Linguistic and Cognitive Engineering (DOLCE), [7]

Ontologies such as these are fairly elaborate things, often beginning with an almost philosophical discussion of types of concepts and their relationships and uses. It is not the purpose of the present compendium to address those issues, and we shall merely make a few notes about the technical and administrative aspects of some of the ontologies, and in particular how they relate to the styles and representation languages that have been described in earlier chapters.

The Protégé Ontology Library [8] contains a considerable number of ontologies, in particular in the two formats that are supported by Protégé, i.e. OWL and OKBC. Contents range from the very general, such as DOLCE, to the quite specific e.g. `Daycare` - "A demo ontology about a childcare

[5] http://en.wikipedia.org/wiki/Cyc
[6] http://en.wikipedia.org/wiki/Suggested_Upper_Merged_Ontology
[7] http://www.loa-cnr.it/Papers/DOLCE2.1-FOL.pdf
[8] http://protegewiki.stanford.edu/wiki/Protege_Ontology_Library

center showing the use of SWRL for reasoning,” or `Camera` - “An OWL ontology about the individual parts of a photo camera.”

### 8.3.1 The SUMO Ontology

SUMO uses the KIF representation language.

*Text to be added.*

### 8.3.2 The DOLCE Ontology

*Text to be added.* The *WonderWeb Foundational Ontologies Library* (WFOL) is intended to contain a variety of ontology documents together with tools for relating them. DOLCE is the first item in that library.

### 8.3.3 The Cyc Ontology

The Cyc ontology uses the CycL representation language.

*Text to be added.*